Improving the Efficiency of Interactive Sequential Pattern Mining by Incremental Pattern Discovery

Ming-Yen Lin and Suh-Yin Lee*
Department of Computer Science and Information Engineering
National Chiao Tung University, Taiwan 30050, R.O.C.
E-mail: {mylin, sylee}@csie.nctu.edu.tw

Abstract

The discovery of sequential patterns, which extends beyond frequent item-set finding of association rule mining, has become a challenging task due to its complexity. Essentially, a user would specify a minimum support threshold with respect to the database to find out the desired patterns. The mining process is usually iterative since the user must try various thresholds to obtain the satisfactory result. Therefore, the time-consuming process has to be repeated several times. However, current approaches are inadequate for such process due to the long execution time required for each trial. In order to minimize the total execution time and the response time for each trial, we propose a knowledge base assisted algorithm for interactive sequence discovery, called KISP. KISP constructs a knowledge base accumulating the pattern information in individual mining, eliminates considerable amount of potential patterns to facilitate efficient support counting, and speeds up the whole process. In addition, we further optimize the algorithm by direct generations of the reduced candidate sets and concurrent counting of variable sized candidates. For some queries, KISP may eliminate database access completely. The conducted experiments show that KISP outperforms GSP, a state-of-the-art sequence mining algorithm, by several orders of magnitudes for interactive sequence discovery.

1. Introduction

Mining sequential patterns, which finds out temporal associations among item-sets in the sequence database, is an important issue in data mining [2, 4, 6, 7, 13, 16]. A classic application of the problem is the market basket analysis whose database contains purchase records, where each record is an ordered sequence of *itemsets* (sets of items) bought by a customer. The objective is to discover the itemsets in future purchase after certain itemsets were bought. For example, we may obtain a sequential pattern "(*1*, *3*, *4*)⇒(*2*, *5*) [support=30%]" after mining. The pattern indicates that 30% of the customers who purchase items *1*, *3* and *4* at the same time would buy items *2* and *5* later. The mining technique can be applied to various domains such as discovering the relationships between the symptoms and certain diseases in medical applications. In comparison to the mining of association rules [3], sequential pattern mining is more complicated because not only the frequent itemsets but also the temporal relationships must be found.

The mining process is very difficult and time-consuming because patterns could be formed by any permutation of itemsets formed by any combination of possible items in the database. In order to distinguish the interesting patterns, a user must supply a *minimum support* threshold (abbreviated *minsup*) for the mining. The result of mining finds out the set of patterns having *supports* greater than or equal to the *minsup*. The *support* of a pattern is the percentage of sequences (in the database) containing the pattern. The discovered patterns are called sequential patterns or frequent sequences. Most approaches focused on minimizing the search space of *potential sequential patterns* (called *candidates*) [2, 14], or on minimizing the required disk I/O due to the multiple database scanning [13, 16]. All these approaches discover the patterns by directly executing the mining algorithms once a *minsup* is specified.

However, the mining process is typically iterative and interactive since a user may specify a *minsup* value that results in too many or too few patterns. Usually, the user must try various *minsups* until the result is satisfactory. Nevertheless, most approaches are not designed to deal with repeated mining under such circumstance so that each *minsup* invokes a re-mining from scratch. Some approaches solved the interactive problem by pre-processing using an assumed least *minsup* [11]. Nevertheless, the lengthy pre-processing has to be executed again if a user supplies a *minsup* below the assumed least value.

In this paper, we propose a simple approach, called *KISP*, to improve the efficiency of sequential pattern discovery with changing supports. *KISP* utilizes the

information obtained from prior mining processes, and generates a knowledge base (abbreviated *KB*) for further queries about sequential patterns of various *minsups*. When the results cannot be directly derived from the knowledge base, *KISP* incorporates *KB* into a fast sequence discovery by eliminating the candidates existing in *KB* before support counting. Unlike those approaches assuming a least *minsup* for pre-processing before iterative mining, *KISP* accepts any *minsup* value and has no difficulty in mining huge databases even with a small main memory. Furthermore, we provide two optimizations to speed up *KISP*. The direct generation of new candidates eliminates candidate searching in *KB* and the concurrent support counting reduces database scanning. The conducted experiments on well-known synthetic data show that the proposed algorithm effectively improves the interactive mining performance.

The rest of the paper is organized as follows. We formulate the problem of interactive sequential pattern mining in Section 2 and review some related algorithms in Section 3. Section 4 presents the proposed approach for the interactive discovery problem. The experimental evaluation is described in Section 5. Section 6 concludes our study.

2. The problem of interactive sequence discovery

Let $\Psi = \{\alpha_1, \alpha_2, \ldots, \alpha_z\}$ be a set of literals, called *items*. An *itemset* $I = (\beta_1, \beta_2, \ldots, \beta_m)$ is a set of m items, such that $I \subseteq \Psi$. A *sequence* x, denoted by $<a_1a_2\ldots a_n>$, is an ordered set of n *elements* where each *element* a_j is an itemset. The *size* of the sequence x, denoted by $|x|$, is the total number of items in all the elements in x. Sequence x is a k-sequence if $|x| = k$. For example, <(2)(3,4)> and <(1)(2)(1)> are all 3-sequences. A sequence $\omega = <b_1b_2\ldots b_w>$ is a *subsequence* of another sequence $\varpi = <a_1a_2\ldots a_n>$ if there exist $1 \le i_1 < i_2 < \ldots < i_w \le n$ such that $b_1 \subseteq a_{i_1}$, $b_2 \subseteq a_{i_2}$, …, and $b_w \subseteq a_{i_w}$. Sequence ϖ *contains* sequence ω if ω is a subsequence of ϖ. For instance, <(2)(5)> is a subsequence of <(2)(1)(3,5)> since (2) $\subseteq$ (2) and (5) $\subseteq$ (3,5).

Each customer record in the database *DB* is referred to as a *data sequence*, which is a sequence of purchased itemsets ordered by transaction time. The *support* of sequence x, denoted by *x.sup*, is the number of data sequences containing x divided by the total number of data sequences in database *DB*. The *minsup* is the user specified minimum support threshold. A sequence x is a *frequent sequence* if $x.sup \ge minsup$. The sequence x is also called a ***sequential pattern***. Given the *minsup* and the database *DB*, the problem of sequential pattern mining is to discover *the set of all sequential patterns*, denoted by S[*minsup*].

The interactive sequence discovery process is described as follows. Given the database *DB*, the user queries with several *minsup* values interactively, and finds out the desired set of sequential patterns with respect to the final *minsup*. The objective of interactive discovery is to respond to each query quickly, and to minimize the overall mining time for the whole process accordingly.

3. Related work

3.1. Algorithms for sequential pattern mining

The *AprioriAll* [2] is the first algorithm dealing with sequential pattern discovery [2, 7, 15]. In subsequent work, the same authors proposed the *GSP* (**G**eneralized **S**equential **P**attern) algorithm that outperforms *AprioriAll* [14]. *GSP* discovers the sequential patterns through multiple database scanning by generating-and-testing candidate k-sequences in k-th database pass. The candidates having enough supports become seeds for generating candidates in the next pass. *GSP* terminates when there is no candidate any more. We further describe the two essential sub-processes in the following.

Candidate generation: Let S_k[*minsup*] be the set of all frequent k-sequences, and X_k[*minsup*] be the set of all candidate k-sequences with respect to *minsup*. *GSP* generates X_k[*minsup*] by self-joining S_{k-1}[*minsup*] to obtain a superset of X_k[*minsup*] and pruning those candidates having any (*k-1*)-subsequence which is not in S_{k-1}[*minsup*]. In the first step, we join a sequence x with another sequence y if the subsequence obtained by dropping the first item of x is the same as the subsequence obtained by dropping the last item of y. The resultant candidate from this join is x extended with the last item of y. For example, the candidate <(1)(3)(5)> is generated by joining <(1)(3)> with <(3)(5)>, and the candidate <(1)(3,5)> is generated by joining <(1)(3)> with <(3,5)>. Moreover, X_k[*minsup*] $\supseteq$ S_k[*minsup*] [14].

Support counting: *GSP* stores candidates in a hash-tree structure [3, 14]. Candidates would be placed in the same leaf if their leading items, starting from the first item, were hashed to the same node. The next item is used for hashing when an interior node, instead of a leaf node, is reached [14]. The candidates required for checking against a data sequence are located in leaves reached by applying the hashing procedure on each item of the data sequence [14]. The support of the candidate is incremented by one if it is contained in the data sequence.

In addition, the *SPADE* (**S**equential **PA**ttern **D**iscovery using **E**quivalence classes) algorithm finds out sequential patterns using vertical database layout and join-operations [16]. Vertical database layout transforms data sequences into item-oriented lists. The lists are joined to form a sequence lattice, in which *SPADE* searches and discovers the patterns [16].

Recently, the *FreeSpan* (**Fre**qu**e**nt pattern-projected **S**equential **Pa**ttern Mining) algorithm was proposed to mine sequential patterns by a database projection technique [4]. Based on a similar projection technique, the authors in [12] proposed the *PrefixSpan* (**Prefix**-projected **S**equential **pa**ttern mining) algorithm. *PrefixSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each projected database. Employing a divide-and-conquer strategy with the *PatternGrowth* methodology, *PrefixSpan* efficiently mines the complete set of patterns [12].

However, these algorithms re-execute the mining procedure every time a new *minsup* is specified during the interactive process. The response time would be longer for queries with smaller *minsup* values.

3.2. Algorithms for interactive pattern discovery

The problem of interactive association discovery was addressed in [1]. The method in [1] preprocesses the data in the transactional database, and stores frequent itemsets in an adjacency lattice. Online repeated queries about association rules are answered by graph theoretic searching on the lattice.

Similarly, a knowledge cache is used for interactive association discovery in [9]. The knowledge cache contains frequent itemsets and the non-frequent itemsets, if memory space is available, that have been discovered while processing other queries. The study [9] indicated that their *benefit replacement* algorithm is the best caching algorithm.

Although on-line association discovery [1, 5, 9, 10] is close to our problem, these approaches aim to interactively find frequent itemsets rather than frequent sequences, which is more complicated. One related work of interactive sequence mining extended the *SPADE* algorithm [16] into the *ISM* (Incremental Sequence Mining) algorithm for incremental and interactive sequence mining [11]. All queries are performed on a pre-processed in-memory data structure, the Increment Sequence Lattice (*ISL*). Therefore, A 'small enough' *minsup* must be pre-selected to apply *SPADE* for pre-processing and saving the results in *ISL*. Nevertheless, if a query involves a threshold smaller than the pre-selected *minsup*, another (more) lengthy mining process must be performed to generate a new *ISL* for the new query. Moreover, as described in [11], the *ISM* might encounter memory problem if the number of the potentially frequent patterns is too large.

Without any assumption on the *minsup* value and on the required memory, the proposed algorithm speeds up interactive sequence discovery by using the acquired information with optimizations like direct candidate-generation and concurrent counting.

4. The proposed algorithm for interactive sequence discovery

4.1. The *KISP* (Knowledge base assisted Incremental Sequential Pattern) mining algorithm

KISP enhances *GSP* with a *knowledge base* (denoted by *KB*) for interactive sequence discovery. When the desired patterns cannot be obtained from *KB*, *KISP* speeds up support counting by eliminating considerable amounts of candidates existing in *KB*. Two optimizations, direct new-candidate generation in Section 4.2 and concurrent support-counting in Section 4.3, further make a significant performance improvement. Figure 1 outlines the proposed *KISP* algorithm for interactive discovery of sequential patterns.

KISP works like *GSP* for the very first mining. The fundamental *KB* is built, only once, by a simple scan over the database to count the supports of candidate *1*-sequences. The supports of all candidate *1*-sequences are included in *KB*, and *S*[*minsup*] contains the frequent *1*-sequences. At the end of this mining, *KB* would collect the supports of all the candidates in each pass, and *KB.base* is the *minsup* designated for this mining.

In addition to the sequential patterns, we also keep the supports of all counted candidates in *KB* regardless of their values for two reasons. First, several currently non-frequent candidates might turn out to be frequent when a smaller *minsup* is specified subsequently. We can immediately obtain these patterns from *KB* without any database access. Second, to find out the true patterns, the mining process generally counts a large number of candidates although they are eventually rejected. We can get rid of the 'useless counting' for the 'commonly non-frequent' candidates if their supports were kept. For example, those candidates ever counted with the support value of zero would not be inserted into the candidate hash-tree afterward. Consequently, a faster counting is enabled due to the smaller hash-tree of the reduced set of candidates.

For subsequent queries, the non-empty *KB* contains the supports of all the generated candidates while mining with *KB.base* as the support threshold. Assume that the user specifies *minsup* for mining. The values of *minsup* and *KB.base* determine whether new counting is required. If the *minsup* is greater than *KB.base*, we simply retrieve from *KB* those patterns satisfying the new *minsup*. *KB* and *KB.base* stay intact since no counting is performed. Tremendous gains in performance can be resulted from direct retrieval of valid patterns from *KB* without re-examining the huge database.

Algorithm KISP (*DB*, *KB*, *minsup*)
Input: *DB* = the database of data sequences; *minsup* = user specified minimum support ;
KB = knowledge base having the supports of all the candidates in prior minings
Output : *S*[*minsup*] = sequential patterns with respect to *minsup*; *KB* = (new) knowledge base

// Let *x.sup* be the support of a candidate *x* , X_k[*minsup*] be the set of candidate *k*-sequence in *DB* with
// respect to *minsup, and KB.base* be the counting base (the smallest *minsup* used) in constructing the *KB*
1) if *KB* = ϕ then *KB* = {*x* and *x.sup*, $\forall\, x \in X_1$} ;
2) *S*[*minsup*] = {*x*| $x \in KB \wedge x.sup \geq minsup$} ; // obtain valid sequential patterns from knowledge base
3) if *minsup* < *KB.base* then // mine new patterns and accumulate new knowledge
4) *k* = 2 ;
5) generate X_k [*minsup*] from the frequent (*k-1*)-sequences in *S*[*minsup*] ;
6) X_k'= X_k [*minsup*] - {*x*| $x \in KB$} ; // eliminate those candidate *k*-sequences in *KB*
7) while $X_k' \neq \phi$ do // there exist candidate *k*-sequences, obtains their supports
8) forall data sequences *ds* in database *DB* do
9) for each candidate $x \in X_k'$ do
10) increase the support of *x* if *x* is contained in *ds* ;
11) endfor
12) endfor
13) *KB* = *KB* ∪ {*x* and *x.sup*, $\forall\, x \in X_k'$} ; // collect new candidates and their supports
14) *S*[*minsup*] = *S*[*minsup*] ∪ {*x* | $x.sup \geq minsup \wedge x \in X_k'$} ; // collect new patterns from X_k'
15) *k* = *k*+1 ;
16) generate X_k [*minsup*] from the frequent (*k-1*)-sequences in *S*[*minsup*] ;
17) X_k'= X_k [*minsup*] - {*x*| $x \in KB$} ; // the reduced set eliminates candidate *k*-sequences in *KB*
18) endwhile
19) *KB.base* = *minsup* ; // update the counting base of *KB*
20)endif

Figure 1. Algorithm *KISP*

In case that *minsup* is smaller than *KB.base*, we have to mine the database for new patterns that are not in *KB*. The fundamental difference between *KISP* and *GSP* is that *KISP* only needs to count the supports of the 'new' candidates by sparing the candidates already existing in *KB*. Even the modest technique spares the counting of a substantial number of candidates, as confirmed by our experiments. In each pass, we first generate the candidates and then remove those existing in *KB*. Next, we expand *KB* with the support of every new candidate to re-use the counting effort for future queries. The sequential patterns are collected for the corresponding query. Finally, *KB.base* is replaced by the new *minsup* since the counting base is changed. The 'new pattern' mining part, which is also the part of new information acquisition step, of the procedure is activated only when the subsequent *minsup* is smaller than *KB.base*.

The counting effort of each mining incrementally expands *KB* so that *KISP* is gradually enhanced with greater candidate reduction capability. *KISP* becomes more powerful as the *minsup* gets smaller gradually during the interactive process.

4.2. New-candidate generation by direct computation

As described in Section 4.1, *KISP* removes the candidates existing in *KB* before counting. The remaining candidates are referred to as *new-candidates*. Instead of generating all candidates and then removing the counted ones, we use Theorem 1 to generate the new-candidates in pass *k* (denoted by X_k') directly. In Theorem 1, S_k[*minsup*] denotes the set of frequent *k*-sequences, X_k[*minsup*] denotes the set of candidate *k*-sequences with respect to *minsup*, and "⊗" represents the join operation described in Section 3.1. We use N_k[*minsup*] to designate the new frequent *k*-sequences (due to *minsup*) by contrast to the frequent *k*-sequences in *KB*. Hence, S_k[*minsup*] = S_k[*KB.base*] ∪ N_k[*minsup*].

Theorem 1. X_k' = (S_{k-1}[*KB.base*] ⊗ N_{k-1}[*minsup*]) ∪ (N_{k-1}[*minsup*] ⊗ N_{k-1}[*minsup*]). That is, X_k' is the union of the two sets; one obtained from joining the frequent (*k-1*)-sequences in *KB* with the new frequent (*k-1*)-sequences, the other obtained from self-joining the new frequent (*k-1*)-sequences.

Proof. By definition, X_k[*minsup*] = S_{k-1}[*minsup*] ⊗ S_{k-1}[*minsup*].

1) X_k[*minsup*] = (S_{k-1}[*KB.base*] ∪ N_{k-1}[*minsup*]) ⊗

(S_{k-1}[*KB.base*] ∪ N_{k-1}[*minsup*]).

2) X_k[*minsup*] = (S_{k-1}[*KB.base*] ⊗ S_{k-1}[*KB.base*]) ∪ (S_{k-1}[*KB.base*] ⊗ N_{k-1}[*minsup*]) ∪ (N_{k-1}[*minsup*] ⊗ S_{k-1}[*KB.base*]) ∪ (N_{k-1}[*minsup*] ⊗ N_{k-1}[*minsup*]).
3) X_k[*minsup*] = X_k[*KB.base*] ∪ (S_{k-1}[*KB.base*] ⊗ N_{k-1}[*minsup*]) ∪ (N_{k-1}[*minsup*] ⊗ N_{k-1}[*minsup*]) due to X_k[*KB.base*] = S_{k-1}[*KB.base*] ⊗ S_{k-1}[*KB.base*] and that N_{k-1}[*minsup*] ⊗ S_{k-1}[*KB.base*] is the same set as S_{k-1}[*KB.base*] ⊗ N_{k-1}[*minsup*].
4) Since $X_k' = X_k$[*minsup*] − X_k[*KB.base*], X_k' = (S_{k-1}[*KB.base*] ⊗ N_{k-1}[*minsup*]) ∪ (N_{k-1}[*minsup*] ⊗ N_{k-1}[*minsup*]). □

4.3. Concurrent support counting and the placement of variable sized candidates

KISP might have only very few new-candidates at a very low *minsup* value since the information gathered from each mining all contribute to the candidate reduction. In each pass, the number of candidates inserted into the hash-tree is far less than that of *GSP*. Thus, *KISP* is enabled to accommodate more candidates in the same hash-tree during the same pass of database scanning. Consequently, we may release the restriction of counting only candidate *k*-sequences to count variable sized candidates concurrently in pass *k*, and reduce the total number of database passes.

KISP assumes nothing about the main memory size so that the available memory might be insufficient for generating new candidate *k*-sequences. Analogous to *GSP*, if the set of frequent (*k-1*)-sequences cannot fit into the memory, we apply the relational merge-join technique without pruning [14] to generate X_k', and then output X_k' to disk after counting.

On the contrary, the set of X_k' is possibly small and occupies only a small part of the memory. We maximize memory utilization by continuously generating the candidates of longer size into the hash-tree until the memory space is nearly full. *KISP* can estimate the space required for X_k' with S_{k-1}[*KB.base*] and N_{k-1}[*minsup*], as described in the following.

Considering the number of candidates generated in each pass, X_2' has more candidates than any other X_k' because none in the candidate superset of size two can be pruned. For candidates of X_k' where $k > 2$, some frequent (*k-1*)-sequences are not to be joined if their subsequences do not match. Assume the number of patterns in S_1[*KB.base*] is p and the number of patterns in N_1[*minsup*] is q. The number of new-candidates in pass 2 is $[3(p+q)^2-(p+q)]/2-(3p^2-p)/2 = 3pq+(3q^2-q)/2$. This formula can be applied to estimate roughly the maximum number of candidates in other passes. Whenever there is room for the next set of candidates in a batch, *KISP* continuously generates and inserts them into the same hash-tree.

The hash-tree in *GSP* is designed to store the same sized candidates in the leaves originally. Accommodating variable sized candidates in the same hash-tree might produce the problem of having no item for hashing. For example, inserting a candidate *4*-sequence might cause the re-distribution of an overflowed node, while the re-distribution might need to hash on the fourth item of a candidate *3*-sequence in the node. We modify the hashing procedure to put the same prefixed candidates, despite their sizes, in the same leaf. In case there is no item for hashing any more, the candidate is stored in one of the descendent leaves (due to splitting the overflowed leaf). We select the leaf having the fewest number of candidates stored to maximize memory utilization. Since candidates of different size are stored in the same hash-tree, we can check the variable sized candidates against a data sequence at the same time. Therefore, the concurrent support counting minimize the number of database scanning required in *KISP*.

Note that a similar technique named *pass bundling* is described for association mining in [8]. However, *pass bundling* statically sets a limit to determine whether the generation should be continued or not, while *KISP* dynamically estimates and computes the available memory for maximum utilization.

4.4. Manipulation of the knowledge base

The knowledge base should provide fast access to the supports of patterns, support quick estimation of candidate storage required, and be able to expand incrementally. Figure 2 displays the logical structure of the knowledge base.

A knowledge base is composed of a **minimal *KB.base***, and one or more ***KB* heads**. The minimal *KB.base* is the smallest *KB.base* among all the *KB.bases* in the *KB* heads. We create a *KB* head to store the newly acquired information only when the 'new pattern' mining part of *KISP* is executed. A *KB* head comprises (1) a ***KB.base*** indicating the counting base while adding this head (2) the **number of pattern-support heads (ps_heads)** indicating the total number of pattern-support heads in this *KB* head (3) the **pattern-support heads** summarizing the pattern-support tables, and (4) the **position of next *KB* head** linking the next *KB* head so that the knowledge base can 'grow' incrementally.

We group all the same sized patterns in the **pattern-support tables** so that the pattern information of desired size can be directly found through the **position of pattern-support** in the corresponding ps_head. The summary information (**the size of the pattern**, the total **number of counted candidates**, and the total **number of non-zero patterns**) in ps_head is used to estimate the number of new-candidates. The pattern-support table is a list of (support, pattern)-pairs. Note that we keep only the

patterns with non-zero support value to minimize the total size of each pattern-support table. The supports of patterns (of the same size) are stored in support-descending order in the structure to ease the searching of valid patterns on answering an online query. An option to eliminate support sorting is writing the supports in the order of hash-tree traversal. Even when the pattern supports are directly stored without sorting, searching within the knowledge base is still more efficient than re-mining.

5. Experimental results

Several experiments were conducted to assess the performance of the *KISP* algorithm, using an 866 MHz Pentium-III PC with 256MB memory running the Windows NT. Like most studies on sequential pattern mining [2, 4, 12, 14, 16], we generated the synthetic datasets for these experiments using the procedure described in [2]. Due to space limit, we only report the results on dataset C*20*-T*2.5*-S*4*-I*1.25* having 100,000 sequences, $|N$=10000, N_S=2500, and N_I=25000. We refer readers to [2] for the details of the parameters.

5.1. Comparisons of *KISP* and *GSP*

The effect of using knowledge base without concurrent counting optimization is studied first. The interactive discovery comprises five consecutive queries, with *minsup* values starting from 2.5% down to 0.5%.

Figure 3 compares the relative performance of *KISP* and *GSP*. The total execution time is 435 minutes by *KISP* and 799 minutes by *GSP*. *KISP* runs faster than *GSP* for individual mining except for the very first mining. Figure 3 also depicts the ratios of the number of candidates in *GSP* to those in *KISP*. Take *minsup* = 0.75% for example, the execution time ratio of *GSP* to *KISP* is 2.1 times. The time saved by *KISP* resulted from the reduced number of candidates—*GSP* counted 2.3 times the number of candidates. To illustrate the accumulating power of *KB*, the number of candidates generated by *GSP* and by *KISP* in each pass is enumerated in Table 1.

KISP exhibits excellent mining capability for query intensive applications. As we increased the number of queries from 3 to 11, the average execution time (also the time required for posterior queries) decreased from 1763 seconds to 514 seconds.

In the experiments with concurrent optimization, the number of database scanning reduced by concurrent support counting is 6, and the reduced execution time is 94 seconds for the mining with *minsup*=0.5%. Most scans were combined in pass three so that the total number of passes and the total execution times were reduced.

When users need to find the appropriate set of patterns by reducing the number of patterns found in a query, the next specified *minsup* would be greater than the counting base of *KB* (*KB.base*). In the next experiment, all *KB.base*s of the *KB*s were 0.5%, and 100 *minsup*s ranging from 0.5% to 2.5% were randomly selected. The mining results are all available in very short time with average execution time 4.3 seconds and maximum execution time 22 seconds. For most queries, the execution time of *KISP* is several orders of magnitude faster than *GSP*, which always re-mines from scratch. However, one drawback of *KISP* is that the size of *KB*, in proportion to the number of candidates stored, might be larger than the size of the original database. For example, the size ratio of *KB* to *DB* is 4.9 for *minsup*=0.5%.

5.2. Scale-up experiments

In the scale-up experiments, the total number of customers was increased from 100K to 1000K, running the same series of *minsup* (2.5% down to 0.5%). Since *KISP* retrieves merely S_{k-1}[*KB.base*] (i.e. frequent (*k-1*)-sequences in *KB*) for generating candidate *k*-sequences, even without large memory, *KISP* may efficiently discover patterns in large databases with *KB*. Figure 4 shows that the execution time of *KISP* increases linearly as the database size increases. The execution times are normalized with respect to the time for 100,000 customers.

6. Conclusions

The knowledge discovering process is iterative and requires many times of mining since no one can predict the best parameters for the desired outcome. Even a change in minimum support value would demand current approaches to execute the time-consuming process again, not to mention the various query operations such as mining constrained patterns [11] or patterns with hierarchy [14].

In this paper, we propose a simple but efficient mining algorithm for interactive discovery of sequential patterns about varying support thresholds. The proposed *KISP* algorithm constructs a knowledge base *KB* in-disk to minimize the response time for iterative mining. No mining is required if the query result is a subset of *KB*; otherwise, we speed up individual mining through accessing only frequent sequences in *KB* for direct new-candidate generation. The proposed approach directly generates only the new candidates not being considered before, concurrently counts variable sized candidates in the same database scanning, and incrementally expands the knowledge base. Only the non-zero patterns grouping by size are kept to minimize the size of *KB* while providing fast access to pattern information. The performed experiments show that *KISP* enhances *GSP* by several orders of magnitude for interactive sequence mining, with good linear scalability.

However, the disk space could be a problem without

further investigation on minimizing *KB* for very low thresholds. Future work may include the maintenance of *KB* for database updating [6]. For interactive queries other than varying thresholds, though we may answer these queries by reading patterns in *KB* into an *ISL*-like [11] pattern-lattice, it is desirable to integrate the query requirements into *KISP* for faster response.

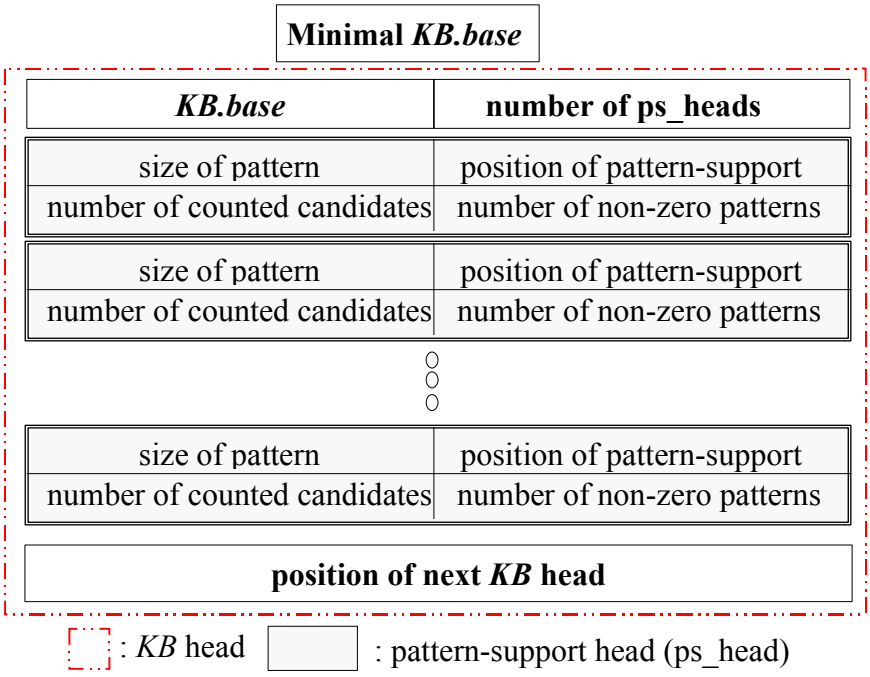

Figure 2. Structure of the knowledge base

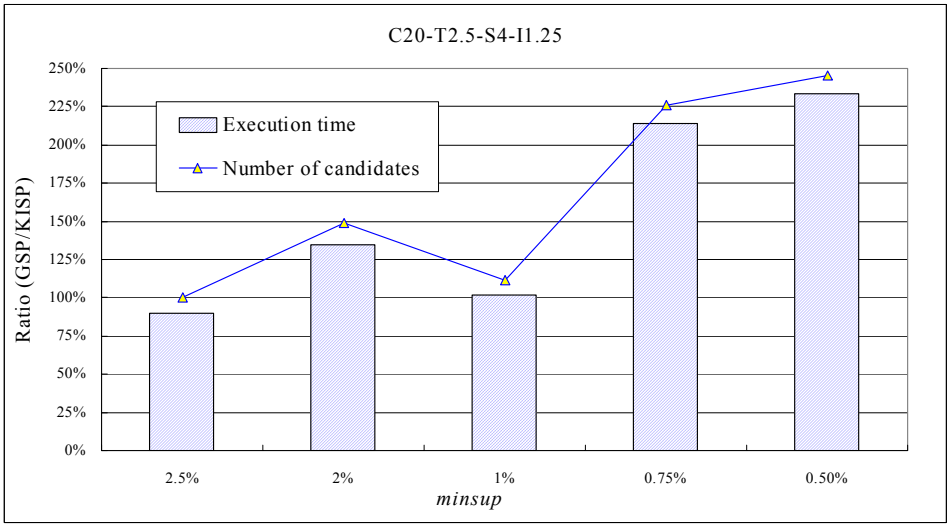

Figure 3. Relative mining performance of *GSP* and *KISP*

Table 1. Number of candidates in each pass (*minsup* =0.5%)

Number of candidates	Pass number							
	1	2	3	4	5	6	7	8
GSP	10000	7673835	7986	2800	1339	430	63	3
KISP	0	3122860	5941	2387	1259	424	63	3

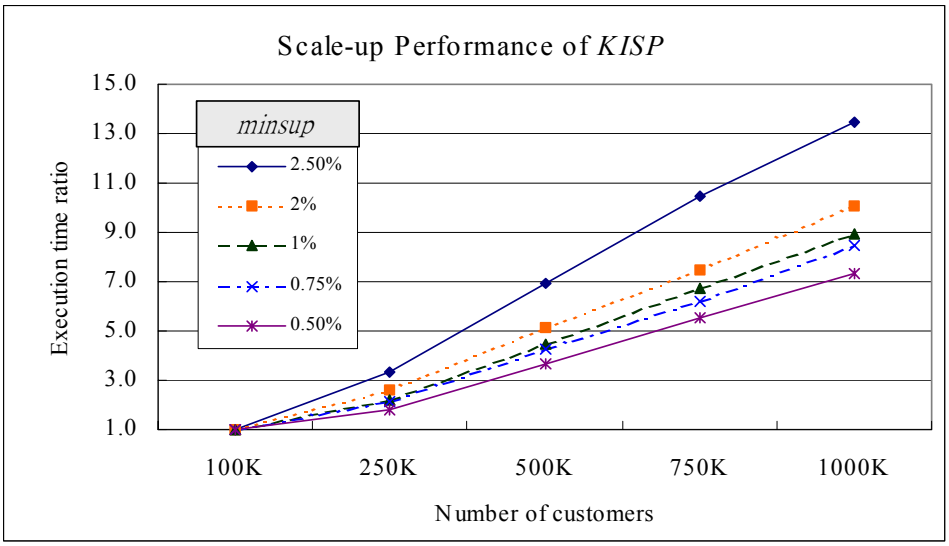

Figure 4. Linear scalability of the database size

Acknowledgements

The authors thank the reviewers' comments for improving the quality of the paper. This research is supported partially by National Science Council of R.O.C. and the LEE and MTI Center for Networking Research at National Chiao Tung Univ., R.O.C.

References

[1] C. C. Aggarwal and P. S. Yu, "Online Generation of Association Rules," *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, USA, Feb. 1998, pp. 402-411.

[2] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, 1995, pp. 3-14.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proceedings of the 20th International Conference on Very Large Data Bases,* Santiago, Chile, Sep. 1994, pp. 487-499.

[4] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal and M.-C. Hsu, "FreeSpan: Frequent pattern-projected sequential pattern mining," *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, pp. 355-359.

[5] C. Hidber, *Online Association Rule Mining*, Technical Report UCB/CSD-98-1004, U. C. at Berkeley, 1998.

[6] M. Y. Lin and S. Y. Lee, "Incremental Update on Sequential Patterns in Large Databases," *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*, 1998, pp. 24-31.

[7] H. Mannila, H. Toivonen and A. I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery*, Vol. 1, Issue 3, 1997, pp. 259-289.

[8] A. M. Mueller, *Fast Sequential and* Parallel *Algorithm for Association Rule Mining: A Comparison*, Technical report CS-TR-3515, University of Maryland, 1995.

[9] B. Nag, P. M. Deshpande and D. J. DeWitt, "Using a Knowledge Cache for Interactive Discovery of Association Rules," *Proceedings of the 1999 SIGKDD Conference*, San Diego, California, Aug. 1999, pp. 244-253.

[10] S. Parthasarathy, S. Dwarkadas and M. Ogihara, "Active Mining in a Distributed Setting," *Proceedings of Workshop on Large-Scale Parallel KDD Systems*, San Diego, CA, USA, Aug. 1999, pp. 65-85.

[11] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas, "Incremental and interactive sequence mining," *Proceedings of the 8th International Conference on Information and Knowledge Management*, Kansas, Missouri, USA, Nov. 1999, pp. 251-258.

[12] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal and M.-C. Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-projected Pattern Growth," *Proceedings of 2001 International Conference on Data Engineering*, 2001, pp. 215-224.

[13] T. Shintani and M. Kitsuregawa, "Mining Algorithms for Sequential Patterns in Parallel: Hash Based Approach," *Proceedings of the Second Pacific–Asia Conference on Knowledge Discovery and Data mining*, 1998, pp. 283-294.

[14] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proceedings* of the *5th International Conference on Extending Database Technology*, Avignon, France, 1996, pp. 3-17.

[15] K. Wang, "Discovering Patterns from Large and Dynamic Sequential Data," *Journal of Intelligent Information Systems*, Vol. 9, No. 1, 1997, pp. 33-56.

[16] M. J. Zaki, "Efficient Enumeration of Frequent Sequences," *Proceedings of the 7th International Conference on Information and Knowledge Management*, Washington, USA, Nov.1998, pp. 68-75.